(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
24 July 2003 (24.07.2003)

PCT

(10) International Publication Number
WO 03/060712 A2

(51) International Patent Classification[7]: G06F 9/46

(21) International Application Number: PCT/US03/01321

(22) International Filing Date: 15 January 2003 (15.01.2003)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:

60/348,566	15 January 2002 (15.01.2002)	US
60/348,707	15 January 2002 (15.01.2002)	US
60/349,344	18 January 2002 (18.01.2002)	US
60/349,424	18 January 2002 (18.01.2002)	US
10/342,113	14 January 2003 (14.01.2003)	US

(71) Applicant *(for all designated States except US)*: **IDETIC, INC.** [US/US]; Suite 510, 2855 Telegraph Avenue, Berkeley, CA 94705 (US).

(72) Inventors; and

(75) Inventors/Applicants *(for US only)*: **DE BONET, Jeremy, S.** [US/US]; 385 Raleigh Tavern Lane, North Andover, MA 01845 (US). **STIERS, Todd, A.** [US/US]; 3208 King Street, Berkeley, CA 94703 (US). **ANNISON, Jeffery, R.** [US/US]; 1585 North Mitchell Canyon Rd., Clayton, CA 94517 (US).

(74) Agent: **MEYER, George, R.**; Gray Cary Ware & Freidenrich, Suite 400, 1221 South MoPac Expressway, Austin, TX 78746-6875 (US).

(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, OM, PH, PL, PT, RO, RU, SC, SD, SE, SG, SK, SL, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, YU, ZA, ZM, ZW.

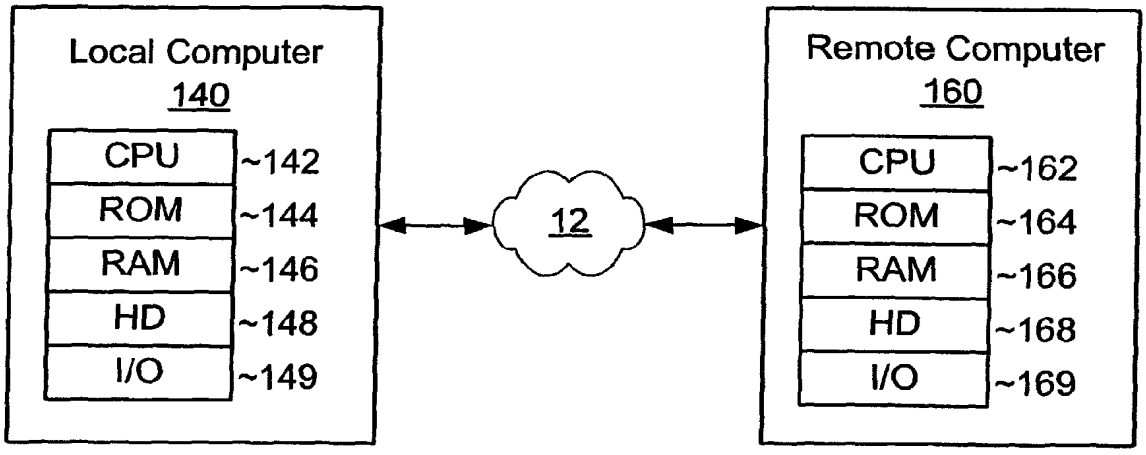
(84) Designated States *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HU, IE, IT, LU, MC, NL, PT, SE, SI, SK, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:

— *without international search report and to be republished upon receipt of that report*

[Continued on next page]

(54) Title: METHOD AND SYSTEM OF ACCESSING SHARED RESOURCES USING CONFIGURABLE MANAGEMENT INFORMATION BASES



(57) Abstract: A method and system can be used to dynamically establish relationships between Simple Network Management Protocol (SNMP) Object Identifiers (OIDs) and data resources with which they are associated. In one embodiment, user-friendly names may be used to identify and access the resources, so that the dynamic association between OID and resource can be achieved by a mapping mechanism that allows for dynamic association between OID and resource name. An OID-name map may be used to link a resource name to one or more OIDs. The map can be generated and modified by altering configuration files, and without having to change software, firmware or hardware. Therefore, generating a new map or modifying an existing map for a new operating environment can be performed quickly and easily.

WO 03/060712 A2

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

METHOD AND SYSTEM OF ACCESSING SHARED RESOURCES USING CONFIGURABLE MANAGEMENT INFORMATION BASES

RELATED APPLICATIONS

This application claims priority under 35 U.S.C. § 119(e) to United States Patent Application Nos. 60/349,424, entitled "Network Proxy Platform that Simultaneously Supports Data Transformation, Storage, and Manipulation for Multiple Protocols" by de Bonet et al., filed on January 18, 2002; 60/349,344 entitled "A Modular Plug-In Transaction Processing Architecture" by de Bonet et al. filed January 18, 2002; and 60/348,566, entitled "Fully Configurable Management Information Bases for the Simple Network Management Protocol" by Annison et al., filed on January 15, 2002. This application is related to United States Patent Application No. (Attorney Docket No. IDET1130-1), entitled "Method And System Of Performing Transactions Using Shared Resources And Different Applications" filed on January 14, 2003 and United States Patent Application No. (Attorney Docket No. IDET1190-1) entitled "Method and System of Protecting Shared Resources Across Multiple Threads" filed on January 15, 2003. All patent applications referenced in this paragraph are assigned to the current assignee hereof and incorporated herein by reference.

REFERENCE TO APPENDICES

Appendices are included in this application by way of attachment, the totality of which is hereby incorporated by reference for all purposes as an integral part of this application. Appendix 1 is entitled "Network Proxy Platform and Its Use" and includes 17 pages of text and 8 sheets of drawings, and Appendix 2 is entitled "Method and System of Protecting Shared Resources Across Multiple Threads" and includes 9 pages of text and 3 sheets of drawings.

FIELD OF THE INVENTION

The invention relates in general to object identifiers, and more particularly, to methods and systems of generating and modifying configurations of management information bases for networks.

DESCRIPTION OF THE RELATED ART

The Simple Network Management Protocol ("SNMP") is the de facto standard used to monitor and manage electronic network infrastructure. SNMP relies on a serial mapping of integers separated by periods (".") to map to a globally specific object. The

objects that SNMP uses are often formatted in a Management Information Base ("MIB"), which like SNMP, is formatted for use in network management software as a map to objects using object identifiers ("OIDs").

Historically, SNMP was designed for use on hardware devices and hard-coded into the firmware of those devices. As network infrastructure became digital and as the Internet began to grow, SNMP became the standard to monitor and manage these networks.

As the networks grew, the use of SNMP spread. With the investments in SNMP monitoring and SNMP-based hardware in place, SNMP began to be used as a tool to not only monitor network nodes (a node can be a participant in a network), but to actually monitor specific pieces of software on those nodes.

The prior limitations of firmware and small footprint devices no longer constrain the ability of the developer to rapidly redesign and redeploy many configurations of an SNMP server (an "agent") in their software. However, the tedious, non-standard nature of connecting SNMP OIDs within modern software is a bottleneck in the process of using and deploying SNMP.

SUMMARY OF THE INVENTION

A method and system can be used to dynamically establish relationships between Simple Network Management Protocol (SNMP) Object Identifiers (OIDs) and data resources with which they are associated. In one embodiment, user-friendly names may be used to identify and access the resources, so that the dynamic association between OID and resource can be achieved by a mapping mechanism that allows for dynamic association between OID and resource name. An OID-name map may be used to link a resource name to one or more OIDs. The map can be generated and modified using software and without having to change firmware or hardware. Therefore, generating a new map or modifying an existing map for a new operating environment can be performed quickly and easily. Systems may not need to be partially or completely shutdown to generate or modify the maps. The method and system are highly beneficial to companies that may incorporate third party subassemblies or code within their products.

In one set of embodiments, a method of accessing a resource can comprise receiving an SNMP-based communication that uses an OID that has been associated with the name of the resource. The method can also comprise determining the resource name that corresponds to the OID used in the request, via an OID-name map. The

method can further comprise determining the data that is associated with that name via a name-resource map and accessing that resource.

In another set of embodiments, a method of using a management information base can comprise generating an OID-name map comprising OIDs and names, wherein each name corresponds to a resource and at least one OID. The method can also comprise modifying the OID-name map using software.

In a further set of embodiments, a system for using a resource can comprise an OID-name map that comprises OIDs and resource names, wherein within the OID-name map, each of the resource names corresponds to at least one OID. The system can also comprise a software component that is configured to receive a communication for an OID, access the OID-name map, determine a resource name that corresponds to the OID, and use that name to retrieve a pointer which can be used to access and manipulate the resource data.

In still further sets of embodiments, data processing system readable media can comprise code that includes instructions for carrying out the methods and may be used on the systems.

The foregoing general description and the following detailed description are exemplary and explanatory only and are not restrictive of the invention, as defined in the appended daims.

BRIEF DESCRIPTION OF THE DRAWINGS

The present invention is illustrated by way of example and not limitation in the accompanying figures.

FIG. 1 includes an illustration of a system comprising two computers coupled to a network that can be used in an embodiment of the present invention.

FIG. 2 includes an illustration of a data processing system storage medium including software code having instructions in accordance with an embodiment of the present invention.

FIG. 3 includes an illustration of a software architecture comprising an application and a resource manager at a local computer, a network, and a remote computer in accordance with an embodiment of the present invention.

FIGs. 4-5 includes a flow diagram of a method of using the resource manager and resource maps in accordance with an embodiment of the present invention.

Skilled artisans appreciate that elements in the figures are illustrated for simplicity and clarity and have not necessarily been drawn to scale. For example, the dimensions

of some of the elements in the figures may be exaggerated relative to other elements to help to improve understanding of embodiments of the present invention.

DETAILED DESCRIPTION

Reference is now made in detail to the exemplary embodiments of the invention, examples of which are illustrated in the accompanying drawings. Wherever possible, the same reference numbers will be used throughout the drawings to refer to the same or like parts (elements).

A method and system can be used to dynamically establish relationships between Simple Network Management Protocol (SNMP) Object Identifiers (OIDs) and data resources with which they are associated. In one embodiment, resource names may be used to identify and access the resources, so that the dynamic association between OID and resource can be achieved by a mapping mechanism that allows for dynamic association between OID and resource name. An OID-name map may be used to link a resource name to one or more OIDs. The map can be generated and modified using software and without having to change firmware or hardware. Therefore, generating a new map or modifying an existing map for a new operating environment can be performed quickly and easily. Systems may not need to be partially or completely shutdown to generate or modify the maps. The method and system are highly beneficial to companies that may incorporate third party subassemblies or code within their products.

As used herein, the terms "comprises," "comprising," "includes," "including," "has," "having" or any other variation thereof, are intended to cover a non-exclusive inclusion. For example, a method, process, article, or apparatus that comprises a list of elements is not necessarily limited to only those elements but may include other elements not expressly listed or inherent to such method, process, article, or apparatus. Further, unless expressly stated to the contrary, "or" refers to an inclusive or and not to an exclusive or. For example, a condition A or B is satisfied by any one of the following: A is true (or present) and B is false (or not present), A is false (or not present) and B is true (or present), and both A and B are true (or present).

The term "software component" is intended to mean at least a portion of a computer program (i.e., a software application). An example includes a software module as used in object-oriented programming. Different software components may reside in the same computer program or in different computer programs on the same computer or different computers.

Before discussing embodiments of the present invention, an exemplary hardware architecture for using embodiments of the present invention is described. FIG. 1 illustrates such an exemplary hardware architecture and includes network 12 bi-directionally coupled to local computer 140 and remote computer 160. Network 12 may be an internal network or an external network (e.g., the Internet). Each of computers 140 and 160 may be a client computer or a server computer, as used in client-server relationships. Note that computers 140 and 160 may also have a peer-to-peer relationship.

Each of computers 140 and 160 can include a server computer, a desktop computer, a laptop computer, a personal digital assistant, a cellular phone, a workstation, or nearly other device capable of communicating over network 12. Other computers (not shown) may also be bi-directionally coupled to network 12. Although the computers are referred to as local computer 140 and remote computer 160, they may be located within separate buildings at distant locations, beside each other in the same room, or anything between those two extremes. In other words, computers 140 and 160 may be considered distinct computers connected to network 12. Alternatively, computers 140 and 160 may represent different portions of a single computer.

In an alternative embodiment, each of local computer 140 and remote computer 160 can be replaced by a plurality of computers (not shown) that may be interconnected to each other over a network or a combination of networks. For simplicity, a single system is shown for each of local computer 140 and remote computer 160.

The local computer 140 can include central processing unit ("CPU") 142, read-only memory ("ROM") 144, random access memory ("RAM") 146, hard drive ("HD") or storage memory 148, and input/output device(s) ("I/O") 149. I/O 149 can include a keyboard, monitor, printer, electronic pointing device (e.g., mouse, trackball, stylus, etc.), or the like. Remote computer 160 can include CPU 162, ROM 164, RAM 166, HD 168, and I/O 169.

Each of the computers in FIG. 1 may have more than one CPU, ROM, RAM, HD, I/O, or other hardware components. For simplicity, each computer is illustrated as having one of each of the hardware components, even if more than one is used. Note that FIG. 1 is a simplification of an exemplary hardware configuration. Many other alternative hardware configurations are possible and known to skilled artisans.

Each of the computers 140 and 160 is an example of a data processing system. ROM 144 and 164; RAM 146 and 166; and HD 148 and 168 can include media that can be read by the CPU 142 or 162. Therefore, each of these types of memories includes a

data processing system readable medium. These memories may be internal or external to the computers 140 or 160.

Portions of the methods described herein may be implemented in suitable software code that may reside within ROM 144 or 164, RAM 146 or 166, or HD 148 or 168. The instructions in an embodiment of the present invention may be contained on a data storage device, such as HD 148. FIG. 2 illustrates a combination of software code elements 204, 206, and 208 that are embodied within a data processing system readable medium 202, on HD 148. Alternatively, the instructions may be stored as software code elements on a DASD array, magnetic tape, floppy diskette, optical storage device, or other appropriate data processing system readable medium or storage device.

In an illustrative embodiment of the invention, the computer-executable instructions may be lines of compiled assembly, C, C^{++}, Java, or other language code. Other architectures may be used. For example, the functions of any one of the computers may be performed by a different computer shown in FIG. 1. Additionally, a computer program or its software components with such code may be embodied in more than one data processing system readable medium in more than one computer.

In the hardware configuration above, the various software components may reside on a single computer or on any combination of separate computers. In alternative embodiments, some or all of the software components may reside on the same computer. For example, one or more the software component(s) of local computer 140 could reside on remote computer 160, or both.

Communications between any of the computers in FIG. 1 can be accomplished using electronic, optical, radio-frequency, or other signals. For example, when a user is at local computer 140, local computer 140 may convert the signals to a human understandable form when sending a communication to the user and may convert input from a human to appropriate electronic, optical, radio-frequency, or other signals to be used by, computers 140 or 160. Similarly, when an operator is at remote computer 160, remote computer 160 may convert the signals to a human understandable form when sending a communication to the operator and may convert input from a human to appropriate electronic, optical, radio-frequency, or other signals to be used by computers 140 or 160.

Attention is now directed to the software architecture of the software in accordance with one embodiment of the present invention. The software architecture enables a dynamic association between OIDs and resources to be achieved by a mapping mechanism that allows for dynamic association between OID and resource name. The software is configured to dynamically reassign OIDs to data within a program.

By configuring the system in one way, a particular OID can be used to access some data, but configured in a different way, a different OID can be used.

The software architecture is illustrated in FIG. 3. Resource manager 700 within local computer 140 may include a name-resource map 200 and an OID-name map 400 that can be used to make calls to resources. Resource manager 700 can include one or more software components to generate, modify, and use maps 200 and 400 when processing communications for shared resources.

The resources may include simple data, such as numbers or character strings, at least part of a structured file (e.g., an eXtensible Markup Language (XML) document), a data file (an image file, audio file, or the like), or the like. Therefore, the types of resources may be highly varied. In one embodiment, a resource for a file system may be used to provide nearly any information regarding the file system including available (free) disk space, a length of an encryption/decryption key, save dates of files, number of read accesses, or nearly any other statistical information that the resource may collect. Resources for other parts of the system may collect and allow access to statistics for those other parts of the systems (e.g., number of users are currently logged onto the system, number of bytes the system has transferred, etc.).

Referring to FIG. 3, application 100 may include any one or more resource names 500, 510, 520, 530, and 540. In one embodiment, resource names 500, 500, 510, 520, 530, and 540 can be user-friendly resource names which aid the programmers when generating code. Each resource name may have a corresponding pointer (300, 310, 320, 330 and 340). Alternatively, the resource name may reference the data itself (also 300, 310, 320, 330 and 340). Map 400 is flexible and can be changed dynamically without having to make any changes to software (other than possibly configuration file(s)), firmware, or hardware. The remote computer 160 may include a program with a function call 1610 for OID 620. The change to map 400 does not affect the code for the program or function call 1610 made by remote computer 160. Further, the modification may even be performed while the program, which has function call 1620, at remote computer 160 is running.

Remote computer 160, such as a network management station, may include the function call 1610 for OID 620 at local computer 140 via network 12. Resource manager 700 may use OID-name map 400 to determine that resource name 500 corresponds to OID 620. Resource manager may use name-resource map 200 to determine pointer 300 corresponds to resource name 500. Pointer 300 may be used to locate the resource used for the function call. Alternatively, reference number 300 may be the data itself, rather than a pointer.

Attention is now directed to FIGs. 4-5 that includes illustrations for a process flow diagram for using shared resources. Referring to FIG. 4, the method can comprise determining names to be used for resources (block 402), generating a name-resource map (block 404), generating an OID-name map (block 406), receiving a call for a resource using the OID (block 422), determining the resource name corresponding to the OID using the OID-name map (block 442), and determining whether the resource has a corresponding pointer (diamond 522 in FIG. 5). If the resource has a corresponding pointer, the method can comprise accessing the resource using the pointer (block 542). Otherwise, the act may be bypassed (i.e., not performed). The method can further comprise sending data over the network in response to the communication (block 562).

Note that not all of the activities described in the process flow diagrams are required, that a limitation within a specific activity may not be required, and that further activities may be performed in addition to those illustrated. Also, some of the activities may be performed substantially simultaneously during with other activities. After reading this specification, skilled artisans will be capable of determining what activities can be used for their specific needs.

Attention is now directed to a more detailed description of the methods as shown in FIGs. 4-5 with references to FIG 3, as appropriate. For the purposes of discussion, local computer 140 comprises all items on the left-hand side of network 12 as shown in FIG. 3, and remote computer 160 comprises a network monitoring station. Remote computer 160 may include a software component that is configured to make function call 1610. Function call 1610 may need to access a resource at local computer 140 corresponding to OID 620 and using network 12.

Before the function call 1610 can be processed at local computer 140, maps and code for the resources need to be in place at local computer 140. In one embodiment, one of the resources may include the available disk space at local computer 140. Therefore, the method can comprise determining names to be used for resources (block 402 in FIG. 4). In one embodiment, a resource that corresponds to available disk space may have a resource name, such as "free_disk_space" instead of using an OID.

After the names are determined, two associative arrays using resource names 500-540 can be generated. A first associative array may include a key of resource name (500-540) and a value of pointer (300-340) that has the address of the resource or it could refer to the resource itself. Therefore, the method can comprise generating name-resource map 200 (block 404), which is the first associative array. A one-to-one relationship may be used for resource name (500-540) to pointer (300-340) though many-to-one mappings are also possible in some embodiments.

In one embodiment, a resource initialization can be performed, for example, using the syntax:

[VARSET] FREE::DISK::SPACE 23.

Resource name 500 is "FREE::DISK::SPACE," and resource 300 has a value of 23. Alternatively, an application program interface ("API") function call could be used to set resource values, for example the function call may be:

bool VarSetOkay = VarSet("FREE::DISK::SPACE",23).

Variable set commands may be used in generating map 200.

The API retrieval from map 200 may be a function call:

int aVariableName = VarGet("FREE::DISK::SPACE",-999).

If –999 is returned, the resource was not found at local computer 140. Otherwise, the value is returned.

A second associative array may include a key of OID (600-650) and a value of resource name (500-540). Alternatively, the name (500-540) may be the key and OID (600-650) may be the value. Therefore, the method can comprise generating OID-name map 400 (block 406), which is the second associative array. Map 400 may be generated potentially by reading in the details from a configuration file or the like.

A configuration function in a configuration file can be used to generate map 400. In one non-limiting embodiment, the format of the command can include:

[DEFINE_OID] <oid> <type> <variable> <readGroup> <writeGroup>,

wherein:

<oid> can be the object identifier;

<type> can be the variable type of the resource name, which will usually be a character string, although other variable types may be used;

<variable> can be the resource name;

<readGroup> can designate who has read access privilege; and

<writeGroup> can designate who has write access privilege.

A non-limiting example may include:

[DEFINE_OID] 1.3.6.1.4.1.11211.3.1.1.1.1 str FREE::DISK::SPACE public operator.

In the non-limiting example, resource name 500 is FREE::DISK::SPACE, and OID 620 is 1.3.6.1.4.1.11211.3.1.1.1.1.

OID-name map 400 and name-resource map 200 may lie within or be accessible to resource manager 700. The resource manager 700 may reside in HD 148 and be loaded into RAM 146 of local computer 140 when it is being used.

The method can comprise receiving a call for a resource using the OID (block 422 in FIG. 4). Referring to FIG. 3, remote computer 160 may include an application having a function call 1610 that is transmitted in a communication using SNMP over network 12 to local computer 140. OID 620, which may have a value of 1.3.6.1.4.1.11211.3.1.1.1.1, may be passed with function call 1610 to obtain available disk space at local computer 140. Function call 1610 may be received by resource manager 700.

The method can also comprise determining the resource name corresponding to the OID using the OID-name map (block 442). Resource manager 700 may access OID-name map 400 to determine that OID 620 (1.3.6.1.4.1.11211.3.1.1.1.1 in the example) corresponds to resource name 500 (FREE::DISK::SPACE in the example).

The method can further comprise determining whether the resource has a corresponding pointer (diamond 522 in FIG. 5). Such a determination may be made using name-resource map 200. If the resource has a pointer, the method can comprise accessing the resource using the pointer (block 542). After obtaining resource name 500 (FREE::DISK::SPACE in the example), resource manager 700 may determine if a corresponding pointer for resource name 500 is within map 200. In this example, resource manager can determine that pointer 300, corresponds to resource name 500. Using pointer 300, resource manager 700 can access the resource requested by function call 1610 (yielding 23 in the example).

In alternative embodiment, reference numbers 300, 310, 320, 330, and 340 may represent data itself rather than a pointer to data. The resources may be accessed directly by using the resource names 500, 510, 520, 530, and 540. In such an embodiment ("No" branch from diamond 522), the method can bypass block 542 in FIG. 5.

The method can comprise sending data over the network in response to the communication (block 562). After accessing the resource, local computer 140 may pass

the requested data, which is the available disk space at local computer 140, back to remote computer 160.

The method and system are powerful in that the software architecture allow for a dynamic reconfiguration between OIDs and resources. The mapping mechanism described above allows for dynamic association between OID and resource name. The software is configured to dynamically reassign OIDs to data within a program. OID-name map 400 and name-resource map 200 may be modified with relative ease. In one embodiment, map 200 or 400 may be modified by changing configuration file(s). The modification can be made without having to change software (other than possibly configuration files), firmware, or hardware. The modification may be made to a map (e.g., configuration file) while a program that accesses a resource within the map is running.

The flexibility and ability to dynamically change OID-name maps without having to change firmware or hardware is highly beneficial when modifying existing configurations or adding new resources. Some hardware may be difficult to reach to replace. In other instances, changing firmware may require shutting down at least part of a system to remove a board to replace a ROM or other similar memory or code. The method and system can obviate the need to partially or completely shutting down a system.

The method and system may also be beneficial to companies that integrate subassemblies from other companies within their own products. The subassemblies can be made that use OIDs that are common with SNMP communications. However, the OID-name map allows the OIDs to be used by the subassembly while other parts of the same product may use the user-friendly names.

The modification may be performed using the configuration commands described previously in this specification. Resources, resource names, OIDs, and pointers may be added, deleted or reconfigured in many different ways. The embodiments described below are non-limiting, illustrative embodiments. Modifications to maps 200 and 400 may be performed using resource manager 700. Within resource manager 700, software component(s) used to modify maps 200 and 400 may be the same or a different software component used to process communications to access resources.

In one embodiment, OID 640 may have originally corresponded to a resource having resource name 550. Data from a resource corresponding to resource name 550 may have been merged into a different resource corresponding to resource name 530. Map 400 can be modified by changing OID 640's corresponding entry in the associative array to replace resource name 550 with resource name 530. The deletion of name 550 its former correspondence with OID 640 is illustrated as dashed lines in FIG. 3. Note that OIDs 610, 630, and 640 map to resource name 530 as illustrated in FIG. 3.

In another embodiment, local computer 140 may be collecting data for its own use corresponding to a resource having resource name 540 and located at the address of pointer 340. Originally, map 400 may not have entries for resource name 540 and OID 650. At a later time, a determination can be made to make the resource at local computer 140 available to remote computer 160. A new pair of entries can be made into the associative array of map 400 to add resource name 540 and OID 650. In this manner, new resources may be easily added at any time.

In still another embodiment, a new resource may be added. Entries for its resource name and point may be added to map 200. Alternatively, a resource may be deleted by removing its name-resource pair from map 200. Modifications to map 400 may or may not be performed in response to modifications to map 200. Therefore, modifications to maps 200 and 400 may occur in a dependent or independent manner.

The system and method allow the use of commands in SNMP to be sent and received in forms that computers use in transferring data over networks. In other words, the get or set commands with OIDs can still be transmitted over a network. However, a receiving computer may take an SNMP-formatted function call and convert it to a format that is used at the receiving computer using a resource name, and potentially a pointer to the resource.

In the foregoing specification, the invention has been described with reference to specific embodiments. However, one of ordinary skill in the art appreciates that various modifications and changes can be made without departing from the scope of the present invention as set forth in the claims below. Accordingly, the specification and figures are to be regarded in an illustrative rather than a restrictive sense, and all such modifications are intended to be included within the scope of present invention.

Benefits, other advantages, and solutions to problems have been described above with regard to specific embodiments. However, the benefits, advantages, solutions to problems, and any element(s) that may cause any benefit, advantage, or solution to occur or become more pronounced are not to be construed as a critical, required, or essential feature or element of any or all the claims.

APPENDIX 1
NETWORK PROXY PLATFORM AND ITS USE

Reference is made in detail to exemplary embodiments of the network proxy platform and its use, examples of which are illustrated in the accompanying drawings. Wherever possible, the same reference numbers will be used throughout the drawings to refer to the same or like parts (elements).

A method and system can comprise a software architecture that allows different applications in the same or different communications protocols to interact with shared resources within a computer. More specifically, code for a computer program may be written to increase the amount of code that is generic to (i.e., shared by) more than one application or communications protocol and reduce the amount of code that handle application-specific or protocol-specific actions. In one embodiment, a transaction may be broken down into a set of discrete actions. The discrete actions may include functions that are common to more than one network application. These functions may be performed by the shared resources.

For each action, code that is specific to a particular protocol or application may be written as part of a software plug-in module with function calls to functions of the shared resources. Each software plug-in module may substantially act similar to a manager for the action, where common tasks are delegated to the shared resources and the module performs specialized functions. Each protocol may have its own set of software plug-in modules for the discrete actions. New applications and support for new protocols can be added by developing a new set of plug-in modules instead of writing an entirely new program. New applications for the same protocol may be developed by replacing or editing as little as one plug-in module from a different application in the same protocol. The software architecture can reduce development time, increase the likelihood that new applications may be developed quickly with fewer changes from an existing application, more protocols will be properly supported, and reduce the burden on hardware and software resources.

A few terms are defined or clarified to aid in understanding the descriptions that follow. A network includes an interconnected set of server and client computers over a publicly available medium (e.g., the Internet) or over an internal (company-owned) system. A user at a client computer may gain access to the network using a network access provider. An Internet Service Provider ("ISP") is a common type of network access provider.

As used herein, the terms "comprises," "comprising," "includes," "including," "has," "having" or any other variation thereof, are intended to cover a non-exclusive inclusion. For example, a method, process, article, or apparatus that comprises a list of elements is not necessarily limited to only those elements but may include other elements not expressly listed or inherent to such method, process, article, or apparatus. Further, unless expressly stated to the contrary, "or" refers to an inclusive or and not to an exclusive or. For example, a condition A or B is satisfied by any one of the following: A is true (or present) and B is false (or not present), A is false (or not present) and B is true (or present), and both A and B are true (or present).

The term "software component" is intended to mean at least a portion of a computer program (i.e., a software application). An example includes a software plug-in module or the like. Different software components may reside in the same computer program or in different computer programs on the same computer or different computers.

Before discussing embodiments of the network proxy platform, an exemplary hardware architecture for using network proxy platform is described. FIG. 1 illustrates such an exemplary hardware architecture and includes client computer 120, proxy computer 140, and server computer 160. Client computer 120 and proxy computer 140 are bi-directionally coupled to network 11, and proxy computer 140 and server computer 160 are bi-directionally coupled to network 13. Each of networks 11 and 13 may be an internal network or an external network (e.g., the Internet). In one embodiment, networks 11 and 13 may be the same network, such as the Internet. Computers 140 and 160 may be bi-directionally coupled to databases 14 and 16, respectively.

Client computer 120 can include a desktop computer, a laptop computer, a personal digital assistant, a cellular phone, or nearly other device capable of communicating over network 11. Other client computers (not shown) may also be bi-directionally coupled to network 11. The proxy computer 140 can be a server computer, but in another embodiment may be a client computer. Other server computers (not shown) similar to server computer 160 may be bi-directionally coupled to network 13.

In an alternative embodiment, each of proxy computer 140 and server computer 160 may be replaced by a plurality of computers (not shown) that may be interconnected to each other over a network or a combination of networks. For simplicity, a single system is shown for each of proxy computer 140 and server computer 160.

The client computer 120 can include central processing unit ("CPU") 122, read-only memory ("ROM") 124, random access memory ("RAM") 126, hard drive ("HD") or storage memory 128, and input/output device(s) ("I/O") 129. I/O 129 can include a keyboard, monitor, printer, electronic pointing device (e.g., mouse, trackball, stylus, etc.),

or the like. Proxy computer 140 can include CPU 142, ROM 144, RAM 146, HD 148, and I/O 149, and server computer 160 can include CPU 162, ROM 164, RAM 166, HD 168, and I/O 169.

Each of the computers in FIG. 1 may have more than one CPU, ROM, RAM, HD, I/O, or other hardware components. For simplicity, each computer is illustrated as having one of each of the hardware components, even if more than one is used. Note that FIG. 1 is a simplification of an exemplary hardware configuration. Many other alternative hardware configurations are possible and known to skilled artisans.

Each of computers 120, 140, and 160 is an example of a data processing system. ROM 124, 144, and 164; RAM 126, 146, and 166; HD 128, 148, and 168; and databases 14 and 16 can include media that can be read by CPU 122, 142, or 162. Therefore, each of these types of memories includes a data processing system readable medium. These memories may be internal or external to computers 120, 140, or 160.

Portions of the methods described herein may be implemented in suitable software code that may reside within ROM 124, 144, or 164, RAM 126, 146, or 166, or HD 128, 148, or 168. The instructions in an embodiment may be contained on a data storage device, such as HD 148. FIG. 2 illustrates a combination of software code elements 204, 206, and 208 that are embodied within a data processing system readable medium 202, on HD 148. Alternatively, the instructions may be stored as software code elements on a DASD array, magnetic tape, floppy diskette, optical storage device, or other appropriate data processing system readable medium or storage device.

In an illustrative embodiment, the computer-executable instructions may be lines of compiled assembly, C, C^{++}, Java, or other language code. Other architectures may be used. For example, the functions of any one of the computers may be performed by a different computer shown in FIG. 1. Additionally, a computer program or its software components with such code may be embodied in more than one data processing system readable medium in more than one computer.

In the hardware configuration above, the various software components may reside on a single computer or on any combination of separate computers. In alternative embodiments, some or all of the software components may reside on the same computer. For example, one or more the software component(s) of the proxy computer 140 could reside on the client computer 120, the server computer 160, or both. In still another embodiment, the proxy computer 140 and database 14 may not be required if the functions performed by the proxy computer 140 are merged into client computer 120 or server computer 160. In such an embodiment, the client computer 120 and server computer 160 may be bi-directionally coupled to the same network (not shown in FIG. 1).

Communications between any of the computers in FIG. 1 can be accomplished using electronic, optical, radio-frequency, or other signals. For example, when a user is at client computer 120, client computer 120 may convert the signals to a human understandable form when sending a communication to the user and may convert input from a human to appropriate electronic, optical, radio-frequency, or other signals to be used by, computers 140 or 160. Similarly, when an operator is at server computer 160, server computer 160 may convert the signals to a human understandable form when sending a communication to the operator and may convert input from a human to appropriate electronic, optical, radio-frequency, or other signals to be used by computers 120, 140, or 160.

Attention is now directed to the methodology of developing a software architecture for the software in accordance with one embodiment. The method can comprise breaking down a transaction into a set of discrete actions. The actual definitions used for separating the transaction into the discrete actions is variable and may be selected by skilled artisans in manners that best suit their particular transactions, hardware requirements, and software requirements. The method can also include determining which functions within the set of discrete actions are common to more than one application. As more are identified, the number of shared resources can increase and the amount of application-specific code can be decreased. Therefore, skilled artisans are encouraged to examine the software from many different levels of abstraction to discover potential shared resources that may otherwise be missed.

The method can further comprise generating software components for the discrete actions. A set of software plug-in modules can correspond to the different discrete actions for the transaction. Each application may have its own set of software plug-in modules. The amount of code within each software plug-in module should be kept relatively low if the identification of shared resources was performed properly. To the extent code for any shared resources does not currently exist, code for the shared resources should be generated to maximize its ability to be used by as many different plug-in modules as possible.

At least two of the software plug-in modules for different applications, whether they use the same or different protocols, can make function calls to any one or more of the shared resources. For different applications using the same protocol, only a request manipulation plug-in module, a content manipulation plug-in module, or both may be the only modules changed. Therefore, creating new application for the same protocol may be simplified because other plug-in modules used for the application may be copied from another application using the same protocol. These other plug-in modules may be

substantially the same between the applications. By replacing or editing the request manipulation plug-in module, content manipulation plug-in module, or both, new applications may be developed very quickly.

Regarding applications in different protocols, each protocol may have a module that performs substantially the same action as any or all of the similar module(s) for the other protocol(s) though reducing this duplicative code by combining the common functionality is preferable.

Attention is now directed to the software architecture of the software in accordance with one embodiment. The software architecture is illustrated in FIGs. 3 and 4 and is directed towards an electronic transaction that can be performed over a network. A basic idea behind the architecture is to allow programming code for shared resources to be commonly used by as many different network applications as possible. Note that all of the resources may or may not be shared by all the applications. The programming code for each application-specific plug-in module may include code to connect the incoming communication in any supported application to the shared resources. By limiting the code within the plug-in modules, a user of the software architecture can reduce development time, increase the likelihood that more applications in the same or different protocols will be properly supported (especially proprietary protocols that may be used by only a limited number of computers or users), and reduce the burden on hardware and software resources for different applications because only relatively small plug-in modules may be used.

In FIG. 3, each row of boxes 3200, 3400, and 3600 represents different applications in the same or different protocols. For example, row 3200 may represent a first application using HTTP, row 3400 may represent a different application using HTTP, and row 3600 may represent yet another application in a different protocol, such as POP, SNMP, WAP, and the like. Note that the series of dots between rows 3400 and 3600 indicate that many other applications in the same or different protocols may be present. Additionally, the architecture may be configured to allow the addition of future applications. The software architecture easily supports at least three different and potentially many more protocols.

Referring to row 3200, each of the boxes 3202 through 3214 represents different stages (actions) that may occur during an electronic transaction. For example, box 3202 may represent a request reception plug-in module, box 3204 may represent an authorization plug-in module, box 3206 may represent a request manipulation plug-in module, box 3208 may represent a content retrieval plug-in module, box 3210 may represents a content manipulation plug-in module, box 3212 may represent a content

delivery plug-in module, and box 3214 may represent a post-response communication plug-in module (e.g., acknowledgement, billing, etc.). Each module may correspond to one or more of the discrete actions. Details about the individual plug-in modules are described later in this specification. Note that the other rows 3400 and 3600 include corresponding boxes for substantially the same types of actions except that they are designed for different applications. More specifically, box 3402 represents an incoming message reception plug-in module for a different application using the same protocol as box 3202, and box 3602 represents an incoming message reception plug-in module for yet another application using a different protocol compared to box 3202.

New applications that make use of already-supported protocols can be developed with a minimum of effort. This is achieved by creating a new row, which makes use of protocol specific plug-ins used in another row and combines them with other plug-ins developed for the specific application at hand. Some plug-in modules may be substantially the same for many different applications in the same protocol. In different protocols, the plug-in modules for at least some of the different applications may provide substantially the same functionality, although the code within those plug-in modules may be different compared to similar modules for the other protocols.

Within the software architecture, shared resources are illustrated as planes 3102, 3104, and 3106 that lie beneath each of the rows 3200, 3400, and 3600. Referring to FIG. 4, interfaces may be made to each of the shared resources for each plug-in module. Specifically referring to box 3214, functional connectivity 4102 links module 3214 and shared resource 3102. Likewise, functional connectivity 4104 links module 3214 and shared resource 3104, and functional connectivity 4106 links module 3214 shared resource 3106. Links 4102, 4104, and 4106 can be achieved by function calls to the shared resources. Examples of the shared resources may include a content cache, a parameter cache, a connection pool, a domain name server cache, a clock, a counter, a database, a global variables space (e.g., a logging database), or the like. A list of potential shared resources is nearly limitless. Note that not all shared resources may be connected to all modules along a row. For example, modules 3202 and 3204 may not need access to the content cache because they may not receive or process content returned for a request. Each connection from a client may be handled independently on its own thread. However in other embodiments, fewer threads or a single thread can be used to operate all connections to a specific row that supports a particular application or protocol. Unless stated to the contrary, the method below is described from the perspective of proxy computer 140.

FIG. 5 includes a flow diagram of a method of performing an electronic transaction that corresponds to the software plug-in modules that lie along any of rows 3200, 3400, and 3600. Note that all modules are not required and that functions of some modules may be combined with others (e.g., authorization may be part of processing an initial request). The process flow diagram will be briefly covered followed by a more detailed description of each module.

The method can comprise receiving a request from a client computer using a request reception plug-in module (block 502) and performing authorization using an authorization plug-in module (block 504). The method can also comprise manipulating a request using a request manipulation plug-in module (block 512). The method can further comprise retrieving content using a content retrieval plug-in module (block 522). The method can yet further comprise manipulating returned content using a content manipulation plug-in module (block 532) and sending the modified content to the client computer using a content delivery plug-in module (block 534). The method can still further comprise processing post-response communications using a post-response plug-in module (block 542).

Note that not all of the activities described in the process flow diagram are required, that a limitation within a specific activity may not be required, and that further activities may be performed in addition to those illustrated. Also, some of the activities may be performed substantially simultaneously during with other activities. After reading this specification, skilled artisans will be capable of determining what activities can be used for their specific needs.

Attention is now directed to the protocol-specific plug-in modules along the rows 3200, 3400, and 3600 and how they are related to the activities illustrated in FIG. 5. Although the discussion is directed to row 3200, the corresponding modules along other rows can provide similar functionality. Also, in the example below, client computer 120 is sending a request for content to proxy computer 140, and server computer 160 is providing content in response to the request. The flow of information could be in the opposite direction (server computer 160 seeking information from client computer 120).

The method can comprise receiving a request from client computer 120 using request reception plug-in module 3202 (block 502 in FIG. 5). Request reception plug-in module 3202 can be used when a request from client computer 120 is received or accessed by proxy computer 140. Module 3202 can initially generate an associative array from portions of the header of the request. Part or all of the associative array may be used by the other modules along the same row. The associative array may provide information that can be part of function calls to the shared resources. Any or all the data

(including the associative array) may be passed from any prior plug-in module (e.g., module 3202) to any or all the subsequent plug-in modules along the same row (e.g., 3204, 3206, 3208, 3210, 3212, or 3214).

The method can also comprise performing authorization using authorization plug-in module 3204 (block 504). The authorization plug-in module 3204 is optional and can be used for determining whether a user at client computer 120 has proper authorization. The authorization modules may be based on an Internet Protocol ("IP") address or a name and a password. Module 3204 may send the IP address or name and password to a shared resource to determine if the user is allowed access.

The method can further comprise manipulating the request using request manipulation plug-in module 3206 (block 512). Request manipulation plug-in module 3206 may be used to modify, replace, or otherwise manipulate the request. For example, proxy computer 140 may have code to redirect a URL within a request to a different URL. More specifically, proxy computer 140 may make a function call to that shared resource using the requested URL. The shared resource may pass the different URL back to module 3206. Module 3206 may have the logic to put the different URL in the correct protocol, so that it will be understood by a computer that may receive the redirected request.

The method can yet further comprise retrieving content using content retrieval plug-in module 3208 (block 522). Content retrieval plug-in module 3208 may be used to send the request and receive or access content in response to the original request or manipulated request. More specifically, a request originating from client computer 120 may have been processed by proxy computer 140 before being received by server computer 160. Content from server computer 160, in response to the processed request from proxy computer 140, would be processed using module 3208. Similar to module 3202, the code may parse the content from server computer 160 into a header portion and a content portion and append that information onto a previously generated associative array.

The method can still further comprise manipulating returned content from the server computer using content manipulation plug-in module 3210 (block 532). Content manipulation plug-in module 3210 may be used to add or modify content before sending it to client computer 120. More specifically, proxy computer 140 may add advertisements or supplementary information from third parties to the content provided by server computer 160. In an alternative embodiment, part or all of the content originating from server computer 160 may be deleted or replaced with other content.

The method can comprise sending the modified content to the client computer using content delivery plug-in module 3212 (block 534). Content delivery plug-in module 3212 may be used to route the content, after manipulation, if any, to client computer 120. Some of the information in the associative array generated when the original request from client computer 120 was processed may be used by module 3212 when sending the outgoing content to client computer 120.

The method can also comprise processing post-response communications using post-response plug-in module 3214 (block 542). Post-response communication plug-in module 3214 may be used for acknowledgement, billing, or other purposes. For example, after content is successfully sent to client computer 120 from module 3212, module 3124 could then charge the user's credit card for that transaction. Alternatively, module 3214 may look for a signal that service to or from client computer 120 or server computer 160 is being terminated for the current transaction. Such post-response processing may be helpful in avoiding invoices or other bills sent to a user at client computer 120 if a product or service was either incomplete or defective or to properly reflect the connect time for a transaction.

Along similar lines, one of the planes as illustrated in FIG. 3 may include global space variables that may need to be used by other shared resources, proxy computer 140, or the plug-in modules. System statistics are examples of information that may be within a global variable space. This information may be useful to proxy computer 140 or another computer, such as client computer 120 or server computer 160, in monitoring activity. The statistics may include how many computers are connected to proxy computer 140, the amount of time each of those computers are connected to proxy computer 140, the amount of or time lapsed during transactions being processed through proxy computer 140, or the like.

These global variables may be used in conjunction with a module, such as authorization module 3204. If too many users are currently logged into proxy computer 140, authorization may be denied even if the computer attempting a connection to proxy computer 140 has proper security clearance. After another transaction by another client computer is terminated, a signal from module 3214 can be sent to the logging system within the shared resources. A new client computer may now gain access to the services provided by proxy computer 140 after the connection from the other transaction is terminated.

Attention is now directed to more specific activities that may be performed by a specific module, and how that specific module may interact with other modules for the same transaction using a specific application. The process flow diagram illustrated in

FIGs. 6-8 is used to describe some of the specific activities. Again, unless stated to the contrary, the method is primarily described from the perspective of proxy computer 140.

To aid in understanding the method in FIGs. 6-8, a specific example is used and occasionally referenced. In the example, an incoming communication may be a request from client computer 120 sent to proxy computer 140 for www.yahoo.com. The client computer 120 is communicating using HTTP, using a Netscape™ browser (of AOL Time Warner, Inc. of New York, New York), and has a MacOS X™ operating system (of Apple Computer, Inc. of Cupertino, California).

Referring to FIG. 6, the method can comprise receiving an incoming communication for a specific application (block 602). The communication can comprise a request, a message, or other form of communication. The communication can be sent by client computer 120 and received or accessed by proxy computer 140 via network 11. Proxy computer 140 can access or read at least a portion of the incoming communication and determine the specific application for the communication. In the example, the incoming communication is a request from client computer 120 sent to proxy computer 140 for www.yahoo.com. The incoming communication will also contain other information within the header of the request. In the example, the other information can include the browser and operating system of client computer 120.

After determining the application for the communication, proxy computer 140 can determine which row 3200, 3400, 3600, or other or row of plug-in modules will be used for the transaction. At this point in the method, proxy computer 140 may activate any or all of the plug-in modules for the row corresponding to the specific application. In one embodiment, plug-in modules within each row may be activated only as they are first used. Referring to the example, the request is for an application corresponding to row 3200. Therefore, plug-in module 3202 may be activated. If the communication is for another application, plug-in module 3402 or 3602 may be activated for the particular application.

The method can further comprise routing the incoming communication to a first software plug-in module for the specific application (block 604). Proxy computer 140 can route the request to request reception software plug-in module 3202 because the incoming request uses the application corresponding to row 3200.

The method can comprise parsing the incoming communication into a header portion and a content portion (block 622). The parsing can be performed by module 3202 to obtain information from the request.

The method can also comprise generating an associative array using information contained within the header portion (block 624). The associative array can include nearly

any finite number of rows. Each row can comprise a key and a value. The key can comprise a parameter within the header portion, and the value can comprise a value for that parameter. In general, the header portion may include one or more lines of a command followed by a command argument. The command may be a key, and the command argument may be the corresponding value for the key. The associative array may be searched by the key or the value.

By knowing conventions used by each of the protocols for incoming communications and the characteristics of headers used for those protocols, formation of the associative array can be performed without complicated coding requirements. The associative array is flexible regarding the number of rows and allows different sizes of associative arrays to be used for different protocols.

For HTTP, one of the lines within the header may include a line with "User-Agent: Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en-US; rv:1.1) Gecko." The key will be "User-Agent," and the value will be " Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en-US; rv:1.1) Gecko." For POP, a line may include "RETR 27," where 27 is an object identifier for is a particular item to be retrieved. The key will be "COMMAND," and the value will be "RETR." A second entry will be made with a key of "ARGUMENT" and a value of "27." For SNMP, a line may include "get 47.12.112.38," where 47.12.112.38 corresponds to an object identifier. The key will be "COMMAND", and the value will be "GET," and a second entry will have the key "ARGUMENT" and the value "47.12.112.38."

The content may or may not be part of the associative array. If it is, the associative array can include a key of "CONTENT" and the entire content data block as the value. For an image, the content may be a very large amount of data. Alternatively, the associative array may be paired with a data pointer that points to the data block, rather than incorporating it directly into the associative array.

Turning to the example, the associative array may include information as shown in Table 1 below. Descriptive names are used instead of actual names to aid in understanding the associative array. Also, the associative array may include many more rows. Because the associative array may be searched by key or value, the order of the rows is unimportant.

Table 1. Exemplary associative array

KEY	VALUE
Request	GET http://www.yahoo.com HTTP/1.0
Request-Document	http://www.yahoo.com
User-Agent	Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en-US; rv:1.1) Gecko
Browser	Netscape
Operating system	MacOS X

The method can also comprise generating a function call to at least one of the shared resources using data within the associative array (block 702 in FIG. 7). In the example, proxy computer 140 can make a function call to a shared resource, more specifically to a clock (shared resource) and a logging system (another shared resource) to get the time and log the beginning of the transaction. The logging information may include the time and a transaction identifier. Note that some of the information within the associative array could be sent with the function call to the shared resource.

The method can further comprise receiving data from the function call (block 704). In the example, the transaction identifier may be passed back to module 3202. The method can still further comprise processing data from the function call with other code within the first software module (block 706). Module 3202 may be more focused on processing the incoming message rather than processing data coming back from the function call. Other modules, such as the content deliver plug-in module 3212, may perform such data processing. Note that the application-specific processing may occur before, during, or after function call(s), if any, are made to the shared resource(s).

A determination may be made whether the first software plug-in module is the last software plug-in module (diamond 722). If so, the method may end. Otherwise, the method may continue with passing any or all of the data (including the associative array) from a prior software plug-in module to the next software plug-in module (block 802 in FIG. 8). In the example, the next software plug-in module is authorization module 3204. Authorization module 3204 may use some of the information that was collected or generated by module 3202. Passing the information reduces the load on hardware by not sending a communication from proxy computer 140 to another computer (e.g., client computer 120) or making the same or similar function call to a shared resource for the same information.

The method can also comprise generating a function call to at least one of the shared resources using data within the associative array (block 822). Authorization

module 3204 may make a function call to the parameter system to determine if the user has proper authorization, whether access can be granted (whether number of users currently connected to proxy computer has exceeded its limits), priority of connection (level or speed of service to be provided), etc. Module 3204 may pass user name and password when making the function call to the logging system. Module 3204 may also make a function call to the shared clock to obtain a time for the action.

The method can also comprise receiving data from the function call (block 824). The data may include information regarding whether user at client computer 120 has proper security clearance, whether the connection could be made, priority of the connection, and the like. The method can further comprise processing data from the function call with other code within the current software plug-in module (block 826). An example may include sending a communication from proxy computer 140 to client computer 120 informing the user whether the connection was made. Alternatively, no further processing may occur with module 3204.

A determination may be made whether the current software plug-in module is the last software plug-in module (diamond 842). If so, the method may end. Otherwise, the method may continue with block 802 in FIG. 8 and proceed in an iterative manner until the last software plug-in module is reached.

The remaining modules along row 3200 will be addressed to complete the example transaction to give a better understanding of actions within the modules and some function calls that those modules may make. More or fewer modules may be used. Also, more, fewer, or different function calls may be made by the modules.

Data can be passed to request manipulation software plug-in module 3206. A function call can be made to a shared resource to determine if the request should be changed. The function call may pass information that a request for www.yahoo.com has been received or accessed. The shared resource may include logic to replace the original client request with www.google.com. The associative array may be changed to replace www.yahoo.com with www.google.com or be appended to note that the manipulated request is www.google.com.

Module 3208 may perform the content retrieval. A function call can be made to a content cache (shared resource) at proxy computer 140 to determine if the content cache includes a network page for www.google.com specifically formatted for a computer having a Netscape™ browser and a MacOS X™ operating system. Note that the browser and operating system information can be obtained from the associative array. If the content cache has the network page, it can be passed to module 3208. Otherwise, module 3208 may formulate an HTTP request to server computer 160 requesting the network page for

the specific browser and operating system of client computer 120. After proxy computer 140 obtains the proper network page from server computer 160, module 3208 may send a function call to the content cache at proxy computer 140 to cache the network page. The proper network page and other information previously collected may be sent to module 3210.

Content manipulation module 3210 may delete, add, or replace some or all of the content within the proper network page returned. For example, when the proper Google network page is received or accessed, module 3210 may add advertisement(s) around the border(s) of the page. A function call can be made to a shared resource to determine which advertisement(s) should be added. The logging system may keep track of which advertisement is being added, whose advertisement it is, and how many times the advertisement has been added during the current billing cycle. The logging system, which is a shared resource, may access the counter (another shared resource) by itself. In other works, some or all of the shared resources may interact with each other without requiring an application-specific software plug-in module to intervene. The manipulated content and other information may be passed to module 3212.

Content delivery software plug-in module 3212 may take the Google network page formatted for a Netscape™ browser and MacOS X™ operating system and the advertisement(s) from module 3210 and prepare a communication using HTTP. The communication can be sent from proxy computer 140 to client computer 120. Function calls can be made to the logging system to note the actual content sent to client computer 120 and time sent. Any or all information collected or generated by modules 3202-3212 may be passed to module 3214.

Post-response communications module 3214 may be used to track usage or billing information. At the end of a transaction, module 3214 may make a function call to the clock to determine the current time, and make another function call to the logging system to determine how much time lapsed during the transaction and record any billing information. The billing information may be within a shared resource managed by an accounting department. Billing information for the user at client computer 120 may be passed from one of the shared resources to module 3214, which may return some of the information for the user at client computer 120. Proxy computer 140 may send a message to client computer 120 similar to "You were connected for 2.1 minutes and were charged $1.27. Thank you for using our service." Alternatively, no message may be sent and the method may end.

Note that not all of the activities described in the process flow diagram in FIGs. 6-8 are required, that a limitation within a specific activity may not be required, and that

further activities may be performed in addition to those illustrated. Also, some of the activities may be performed substantially simultaneously during with other activities. After reading this specification, skilled artisans will be capable of determining what activities can be used for their specific needs.

The power of creating new applications for the same protocol may be better understood with the flow diagram in FIG. 9 and an example. In one embodiment, different applications may be generated for different priorities of users for a network site. The communication protocol may use HTTP. The method can comprise developing a first set of plug-in modules for a first application (block 902). The set may correspond to row 3200 and be directed to premium users of a network site.

A new application may need to be developed for regular users of the network site. The communication protocol may also use HTTP. The method can comprise copying the first set of plug-in modules to form a second set of plug-in modules (block 922).

For the new application, only the request manipulation plug-in module, the content manipulation plug-in module, or both may be replaced. The remainder of the plug-in modules may be unchanged and be substantially the same as the remainder of the plug-in modules for the first application.

The method may comprise replacing a first request manipulation plug-in module with a second request manipulation plug-in module for a second application (block 924). For example, the premium user may have access to some network pages that the regular user may not. If the regular user requests a premium page, the second request manipulation module may direct the regular user to another network page for which the regular user has proper access.

The method may also comprise replacing a first content manipulation plug-in module with a second content manipulation plug-in module for the second application (block 926). The premium user may have only 10 percent of his or her window occupied by advertisements, whereas the regular user may have 50 percent of his or her window occupied by advertisements. The second content manipulation module may reformat the retrieved content to allow for more advertising space. The second content manipulation module may also access the shared resources to obtain the advertisements and keep track of which advertisements were used. Device dependent optimization of network pages (desktop computer vs. cellular phone, etc.) can be achieved by plugging in a module which transcodes content using settings developed for the particular device that made the initial request.

After one or both of the request manipulation and content manipulation modules are replaced, the method can still further comprise executing the second application using the second set of plug-in modules (block 942).

Note that while the example focused more on replacing specific modules, in other embodiments, those modules may be generated by editing code within the corresponding modules within the first set for the first application.

After reading this specification, skilled artisans will appreciate that entirely different applications, using the same network protocol, can be developed by simply inserting new plug-in module(s) at the request manipulation location, the content request location, or both locations.

In other embodiments, the method and system may be used for nearly any other network communications. As an example, client computer 120 may make a request for information within a database located at server computer 160. The request may be handled in a manner similar to a request for a network page. If the user does not have proper authorization to all information within a request, the request manipulation module may request only that information for which the user has property access or the content manipulation module may add information stating that the user does not have proper access to some or all the information.

In another embodiment, the multiple-protocol software architecture and plug-in modules may be installed in client computer 120 or server computer 160. Not all modules in proxy computer 140 may be needed by client computer 120 or server computer 160. Authorization modules 3204, 3404, and 3604 may not be used or can be coded to allow authorization (always authorized) at client computer 120. The content manipulation modules 3210, 3410, and 3610 may not be used by the server computer 160. After reading this specification, skilled artisans are capable of determine which modules are needed and which ones can be eliminated or bypassed (module exists but passes information through without performing any other significant activity).

The software components can be designed to maximize their ability use shared resources while minimizing the amount of code used for application-specific operations. Therefore, relatively smaller plug-in modules (compared to the shared resources) may be used to access the shared resources illustrated in the planes below the modules. In this manner, less code needs to be written for a new protocol compared to the prior-art method of writing or copying and modifying an entire program for a specific protocol. For applications in the same protocol, the specific coding requirements may be much less. Furthermore, protocols are more likely to be supported because the coding requirements are less, and therefore, may be generated for protocols that have relatively fewer users

compared to other protocols. The method and system are significantly more efficient in both time and cost compared to existing prior-art methods dealing with the problem of many different applications in the same or different protocols.

Benefits, other advantages, and solutions to problems have been described above with regard to specific embodiments. However, the benefits, advantages, solutions to problems, and any element(s) that may cause any benefit, advantage, or solution to occur or become more pronounced are not to be construed as a critical, required, or essential feature or element of any or all the claims.

APPENDIX 2
METHOD AND SYSTEM OF PROTECTING SHARED RESOURCES ACROSS MULTIPLE THREADS

Reference is now made in detail to the exemplary embodiments, examples of which are illustrated in the accompanying drawings. Wherever possible, the same reference numbers will be used throughout the drawings to refer to the same or like parts (elements).

Embodiments provide a software system in which a resource access manager prevents the inadvertent corruption of resources shared by multiple threads. This resource access manager provides thread synchronization mechanisms for a shared resource to eliminate the possibility of the resource being corrupted when simultaneously accessed by a plurality of program threads.

In some embodiments access management for a plurality of resources can be controlled by a single multiple resource manager. The multiple resource manager is particularly well suited for computer programs which support the integration of software plug-in modules, potentially written by third-parties.

In some embodiments, read, write, and adjustment operations are forced to occur synchronously. In other embodiments, more sophisticated thread synchronization techniques can be used which can allow read-only operations to occur simultaneously.

As used herein, the terms "comprises," "comprising," "includes," "including," "has," "having" or any other variation thereof, are intended to cover a non-exclusive inclusion. For example, a method, process, article, or apparatus that comprises a list of elements is not necessarily limited to only those elements but may include other elements not expressly listed or inherent to such method, process, article, or apparatus. Further, unless expressly stated to the contrary, "or" refers to an inclusive or and not to an exclusive or. For example, a condition A or B is satisfied by any one of the following: A is true (or present) and B is false (or not present), A is false (or not present) and B is true (or present), and both A and B are true (or present).

The term "software component" is intended to mean at least a portion of a computer program (i.e., a software application). An example includes a software module as used in object-oriented programming. Different software components may reside in the same

computer program or in different computer programs on the same computer or different computers.

Before discussing embodiments, an exemplary hardware architecture for using embodiments is described. FIG. 1 illustrates such an exemplary hardware architecture and includes computer system 100 comprising central processing unit ("CPU") 122. CPU 122 may comprise read-only memory ("ROM"), random access memory ("RAM"), or other types of volatile or non-volatile memory. CPU 122 is bi-directionally coupled to monitor 142, keyboard 144, hard disk ("HD") 162, and printer 164. An electronic pointing device, such as mouse 146, may be coupled to CPU 122 directly (not shown) or via keyboard 144. Other electronic pointing devices can include a trackball, stylus, or the like and may replace or be used in conjunction with mouse 146.

Note that FIG. 1 is a simplification of an exemplary hardware configuration. Computer system 100 may have more than one of hardware components shown in FIG. 1. In addition, other peripheral devices (not shown) may be coupled to CPU 122 or other portion(s) of the computer system 100. Many other alternative hardware configurations are possible and known to skilled artisans.

CPU 122 is an example of a data processing system. HD 162, ROM, RAM, and other memories can include media that can be read by the CPU 122. Therefore, each of these types of memories includes a data processing system readable medium.

Portions of the methods described herein may be implemented in suitable software code that may reside within HD 162, ROM, RAM, or other memory. The instructions in an embodiment may be contained on HD 162 or other memory. FIG. 2 illustrates a combination of software code elements 204, 206, and 208 that are embodied within a data processing system readable medium 202 on HD 162. Alternatively, the instructions may be stored as software code elements on a DASD array, magnetic tape, floppy diskette, optical storage device, or other appropriate data processing system readable medium or storage device.

In an illustrative embodiment, the computer-executable instructions may be lines of compiled assembly, C, C^{++}, Java, or other language code. Other architectures may be used. A computer program or its software components with such code may be embodied in more than one data processing system readable medium in more than one computer.

Communications using computer system 100 in FIG. 1 can be accomplished using electronic, optical, radio-frequency, or other signals. For example, when a user is at computer system 100, CPU 122 may convert the signals to a human understandable form when sending a communication to the user and may convert input from the user to appropriate electronic, optical, radio-frequency, or other signals to be used by, other computer systems (not shown).

Attention is now directed to the software architecture of the software in accordance with one embodiment. The software architecture is illustrated in FIG. 3. A basic idea behind the architecture is to have a resource manager control the access which multiple program threads have to shared resources to prevent data corruption due to concurrent access. The shared resources may include a clock, a counter, a character string, a database, a structured file (an extensible markup language document), an unstructured file (an image file, audio file, or the like), or the like. Therefore, the types of shared resources may be highly varied.

Turning now to FIG. 3 a resource access manager is depicted. Often times, computer programs contain different threads. These threads, which usually execute simultaneously, need access to resources such as those described above to perform properly. However, the shared access these threads have to these resources can cause a myriad number of problems, the most pertinent of which is data corruption and deadlock. Consequently, an embodiment provides a resource access manager 300 which prevents the inadvertent corruption of resources shared by multiple threads. This resource access manager 300 provides thread synchronization mechanisms for a shared resource 310 to eliminate the possibility of the resource being corrupted when simultaneously accessed by a plurality of program threads. Further, this resource access manager is designed so that deadlock cannot occur.

A shared resource 310 may be a clock, counter, data, character strings, clocks, data objects, and the like needed by a thread, or by multiple threads or applications. However, as elaborated on above, when threads simultaneously access this shared resource 310, corruption can occur. To combat this problem, a resource access manager 300 is associated with a shared resource 310.

Resource access manager 300 receives a request from a thread that wishes to access shared resource 310 associated with resource access manager 300. Resource access manager 300 controls this access to shared resource 310 through use of thread

synchronization primitive 320, examples of which are a mutex and a semaphore.

In some embodiments, a mutex can be used by resource access manager 300 to control both read and write access to a shared resource 310. A mutex is a thread synchronization object provided by most modern operation systems, which allows only one thread to lock the mutex at any given time. If a thread tries to lock a mutex which is currently locked, commonly known as "owned" by some other thread, the thread blocks until the mutex is unlocked. Whenever a thread requests access to the resource, via any of the access functions, the thread attempts to lock the resource's mutex. If the resource is currently being accessed, then the thread blocks until the resource is available again. When no other thread is currently accessing the resource, the lock is granted, the access function is performed, and then the mutex is unlocked. This embodiment has the advantage of being simple to implement, however, only one thread may access the resource at a time, even under conditions when several threads could safely have access to the resource simultaneously. A flow diagram for a mutex can be described as:

START: Access request from some thread

|

Is mutex locked?

| '--(yes)-> block until_free

|

(no)

|

Lock mutex

|

Perform access function

|

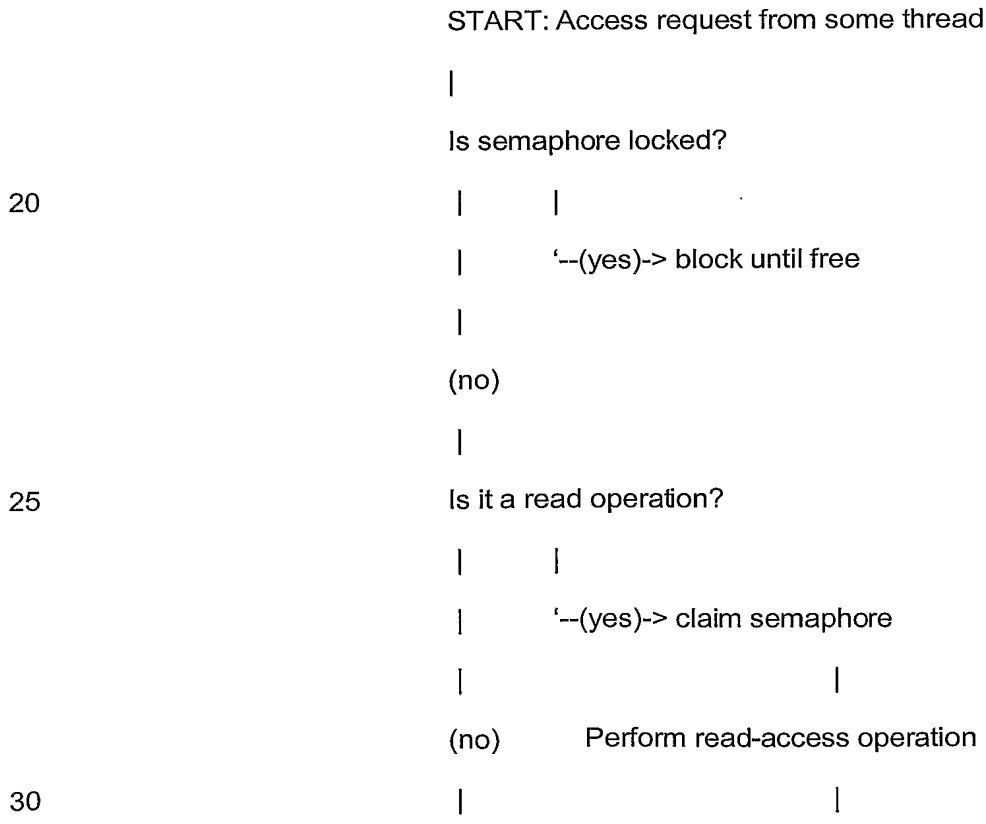
Unlock mutex

|

END: Access completed

In other embodiments, semaphores can be used by resource access manager 300 to control read- and write-access. In addition to the lock/unlock operation supported by mutexes, semaphores support an additional claim/release operation. Any number of threads can have claims to a semaphore, however, no thread can lock a semaphore until all claims have been released. Likewise, no thread can obtain a claim to a semaphore while it is locked. Finally, just like a mutex, only one thread can lock a semaphore at a time.

Thus, when using semaphores to control access to shared resource 310, resource access manager 300 may issue claims to the semaphore associated with a resource to any number of threads, but no more than one thread can lock a resource's semaphore at any time. When a thread attempts to perform a read-operation, an attempt is made to gain a claim to the resource's semaphore. When the read-operation has completed the claim is released. When a thread attempts to perform a write operation, the thread is blocked until there are no claims pending, at which point the write operation can take place. This embodiment has the advantage that multiple threads may access the resource at a time. A flow diagram for a semaphore can be described as:



```
|                release semaphore
|                                |
|                        goto END
|
Is semaphore claimed?
|        |
|        '--(yes)-> block until no claims
|
(no)
|
Lock Semaphore
|
Perform write-access function
|
Unlock Semaphore
|
END: Access completed
```

Resource access manager 300 then allows the calling thread to access shared resource 310 according to the protocol dictated by thread synchronization primitive 320.

Utilizing a resource access manager 300 allows controlling access to shared resource 320 to be moved out of a computer program or thread and into resource access manager 300. To facilitate this process, resource access manager 300 and thread synchronization primitive 320 may be part of a class as that term is used in object-oriented programming. This class may be a class within a library of classes that are provide to programmers, so that a programmer does not need to write his or her own code any time a call to a shared resource is made. For example,

```
#include "shared_res_mgr.h"
```

may be used to invoke this class.

Instead of functions, such as "cin" and "cout" as with the with the C++ programming language's iostream class, the functions in the resource access manager class may comprise "install," "uninstall," "get," "set," "adjust," and "unset." The class is particularly well suited for networked computers that communicate using Simple Network Management Protocol. Note that other names may be used for the functions, such as "retrieve" for "get," "delete" for "unset," and the like.

The functions performed by resource access manager 300 may be classified as read operations, write operations, or a combination of read and write operations. A read operation may be triggered by a "get" command. A write operation may be triggered by an "install", "uninstall", "set", or "unset" command. An "adjust" command may include both a read operation and a write operation.

The "install" and "uninstall" functions may be used for adding or deleting a shared resource 310. The "get" function may be used to read or otherwise obtain a value or other data from shared resource 310. The "set" function may be used to initialize or reset shared resource 310 to a particular value. In one embodiment, the "set" function may set a counter to a value regardless of what value was previously in the counter. For example, a set function of "27" to a counter may set the counter to 27 regardless of the value that was previously in the counter

The "unset" function may delete the data in shared resource 310. The "adjust" function can be similar to the "set" function but change data on a relative basis based on the pre-existing value(s) in shared resource 310. For example, an "adjust" function of "+ 1" sent to a counter may increment the pre-existing value in the count by one. If the counter already had a value of 27 before the adjust function, the value after the adjust function would be 28.

Moving on to FIG. 4, one embodiment which manages access to multiple shared resources is depicted. In one embodiment, the access management for a plurality of resources can be combined into a single multiple resource manager 430. In this particular embodiment, access to any shared resource 416 418, 420 managed by resource manager 430 can be requested by any thread 400, 402, 404, 406 at any time. When access is requested the individual resource access manager 462, 464, 466 for that shared resource 416, 418, 420 controls the access to that resource 416, 418, 420. This arrangement allows for the creation, removal, or replacement of resources at any time by

any thread. Further, it provides a mechanism for software plug-in modules to gain access and share resources without modification of the original program.

Threads 400, 402, 404, 406 of a computer program or application send out access requests 408, 410, 412, 414 for various shared resources 416, 418, 420. These access requests 408, 410, 412, 414 can be routed to multiple resource manager 430. Occasionally, the access requests 408, 410, 412, 414 by threads 400, 402, 404, 406 or programs cannot be handled simultaneously by the multiple resource manager 430, for example multiple "install" or "uninstall" calls. In these cases, access to multiple resource manager 430 must itself be controlled. Controlling access to multiple resource manager 430 is done by associating a resource access manager 432 with the entire multiple resource manager 430. In this manner, access to multiple resource manager 430 can be regulated according to the protocol dictated by the thread synchronization primitive 434 utilized by resource access manager 432 as described above.

Multiple resource manager 430 then receives the resource access requests 408, 410, 412, 414 issued by threads 400, 402, 404, 406. Typically, because shared resources 416, 418, 420 are utilized and referred to in many threads and programs, individual resources are referred to by an identifier, which may be a name, number, pointer or other mechanism. Consequently, multiple resource manager 430 may contain lookup table 440, commonly implemented as an associative array, trie, hash table, or the like, which links an identifier 442, 444, 446 of a shared resource to a pointer 452, 454, 456.

Each pointer 452, 454, 456 indicates the location of the resource access manager 462, 464, 466 associated with the shared resource 416, 418, 420, referred to by the identifier 442, 444, 446. Once the resource access manager 462, 464, 466 for the shared resource 416, 418, 420 is located, a resource access request 408, 410, 412, 414 regarding that shared resource 416, 418, 420 can be routed to the resource access manager 462, 464, 466, for that particular shared resource 416, 418, 420. This resource access request 408, 410, 412, 414, can then be handled by that individual resource access manager 462, 464, 466, according to the corresponding thread synchronization primitive 472, 474, 478 as described above.

The methods and systems described have many advantages over conventional methods and systems. More particularly, the code necessary for performing the function calls as described may be part of a class located within a library of classes available to users of programming languages, such as C, C^{++}, and the like. Therefore, programmers in those

languages who use these classes may not need to know and understand all of the inner workings necessary to properly synchronize and perform function calls in an orderly manner to eliminate the possibility of data corruption or deadlock when multiple threads from different applications are attempting to use the same resource during the same time period.

Benefits, other advantages, and solutions to problems have been described above with regard to specific embodiments. However, the benefits, advantages, solutions to problems, and any element(s) that may cause any benefit, advantage, or solution to occur or become more pronounced are not to be construed as a critical, required, or essential feature or element of any or all the claims.

WHAT IS CLAIMED IS:

1. A method of using a system having a resource, wherein the method comprises:
receiving a communication for a first resource, wherein the communication includes a first OID of the first resource; and
determining a first name of the first resource corresponds to the first OID using an OID-name map; and
accessing the first resource after determining the first name of the first resource.

2. The method of claim 1, further comprising determining a first pointer of the first resource corresponds to the first name using a name-resource map, wherein accessing the first resource comprises accessing the first resource using the first pointer.

3. The method of claim 2, wherein the name-resource map further comprises:
other names; and
other pointers, wherein each of the other pointers corresponds to only one of the other names.

4. The method of claim 2, further comprising modifying the name-resource map.

5. The method of claim 1, wherein the communication is received over a network using SNMP.

6. The method of claim 1, wherein the OID-name map further comprises:
other OIDs for other resources; and
other names, wherein each of the other names corresponds to at least one of the first and other OIDs.

7. The method of claim 6, wherein more than one OID corresponds to the first name.

8. The method of claim 1, further comprising sending a datum over a network in response to the communication, wherein:
accessing the first resource comprises accessing the first resource for the datum; and
sending is performed after accessing the first resource.

9. The method of claim 1, further comprising modifying the OID-name map.

10. The method of claim 1, wherein modifying the OID-name map is performed while a program that accesses the first resource is running.

11. A method of using a management information base comprising:
generating an OID-name map comprising OIDs and names, wherein each name corresponds to a resource and at least one OID; and
modifying the OID-name map using software.

12. The method of claim 11, wherein modifying comprises modifying the OID-name map, such that:
before modifying, a first OID corresponds to a first name; and
after modifying, the first OID corresponds to a second name different from the first name.

13. The method of claim 11, wherein modifying comprises modifying the OID-name map, such that:
before modifying, a first name corresponds to a first OID; and
after modifying, the first name corresponds to a second OID different from the first OID.

14. The method of claim 11, wherein modifying comprises deleting a name or an OID from the OID-name map.

15. The method of claim 11, wherein modifying comprises adding a name or an OID to the OID-name map.

16. The method of claim 11, wherein modifying is performed without making any firmware or hardware change.

17. The method of claim 11, wherein modifying the OID-name map is performed while a program that accesses the first resource is running.

18. A data processing system readable medium having code embodied therein, the code comprising:
an instruction for accessing a communication for a first resource, wherein the communication includes a first OID of the first resource; and
an instruction for using an OID-name map to determine a first name of the first resource corresponds to the first OID; and
an instruction for accessing the first resource after determining the first name of the first resource.

19. The data processing system readable medium of claim 18, wherein:
the code further comprises an instruction for determining a first pointer of the first resource corresponds to the first name using a name-resource map; and
the instruction for accessing the first resource comprises an instruction for accessing the first resource using the first pointer.

20. The data processing system readable medium of claim 19, wherein the name-resource map further comprises:
other names; and
other pointers, wherein each of the other pointers corresponds to only one of the other names.

21. The data processing system readable medium of claim 19, wherein the code further comprises an instruction for modifying the name-resource map.

22. The data processing system readable medium of claim 18, wherein the communication uses SNMP.

23. The data processing system readable medium of claim 18, wherein the OID-name map further comprises:
other OIDs for other resources; and
other names, wherein each of the other names corresponds to at least one of the first and other OIDs.

24. The data processing system readable medium of claim 23, wherein more than one OID corresponds to the first name.

25. The data processing system readable medium of claim 18, wherein:
the code further comprises an instruction for sending a datum over a network in response to the communication;
the instruction for accessing the first resource comprises an instruction for accessing the first resource for the datum; and
the instruction for sending is executed after the instruction for accessing the first resource.

26. The data processing system readable medium of claim 18, wherein the code further comprises an instruction for modifying the OID-name map.

27. The data processing system readable medium of claim 26, wherein the instruction for modifying is configured to allow execution of the modification while a program that accesses the first resource is running.

28. A data processing system readable medium having code embodied therein, the code comprising:
an instruction for accessing an OID-name map comprising OIDs and names, wherein each name corresponds to a resource and at least one OID; and
an instruction for modifying the OID-name map using software.

29. The data processing system readable medium of claim 28, wherein the instruction for modifying comprises an instruction for modifying the OID-name map, such that:
before executing the instruction for modifying, a first OID corresponds to a first name; and
after executing the instruction for modifying, the first OID corresponds to a second name different from the first name.

30. The data processing system readable medium of claim 28, wherein the instruction for modifying comprises an instruction for modifying the OID-name map, such that:
before executing the instruction for modifying, a first name corresponds to a first OID; and
after executing the instruction for modifying, the first name corresponds to a second OID different from the first OID.

31. The data processing system readable medium of claim 28, wherein the instruction for modifying comprises an instruction for deleting a name or an OID from the OID-name map.

32. The data processing system readable medium of claim 28, wherein the instruction for modifying comprises an instruction for adding a name or an OID to the OID-name map.

33. The data processing system readable medium of claim 28, wherein the instruction for modifying is performed without making any firmware or hardware change.

34. The data processing system readable medium of claim 28, wherein the instruction for modifying is configured to allow execution of the modification while a program that accesses the first resource is running.

35. A system for using a resource comprising:
an OID-name map that comprises OIDs and resource names of resources, wherein each of the resource names corresponds to at least one OID;
a first software component that is configured to:
receive a communication for a first OID of a first resource;
access the OID-name map; and
determine a first resource name that corresponds to the first OID.

36. The system of claim 35, further comprising a name-resource map that comprises the resource names and pointers that correspond to the resource names, wherein the first software component is further configured to:
access the name-resource map; and
determine a first pointer of a first resource that corresponds to the first resource name.

37. The system of claim 35, wherein:
more than one OID refers the first resource name; and
each resource name refers to only one resource.

38. The system of claim 35, wherein the communication is in SNMP.

39. The system of claim 35, further comprising a second software component is configured to modify the OID-name map.

40. The system of claim 39, wherein modification of the OID-name map is performed without changing firmware or hardware.

41. The system of claim 39, wherein the second software component is configured to allow modification of the OID-name map while a program that accesses the first resource is running.

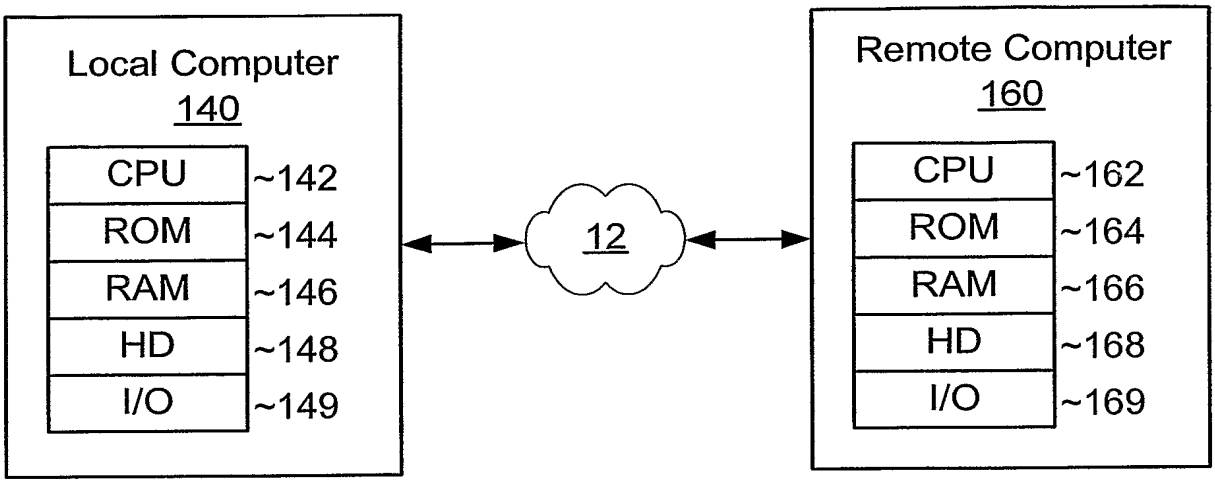

Local Computer
140
CPU
~142
ROM
~144
RAM
~146
HD
~148
I/O
~149
12
Remote Computer
160
CPU
~162
ROM
~164
RAM
~166
HD
~168
I/O
~169

FIG. 1

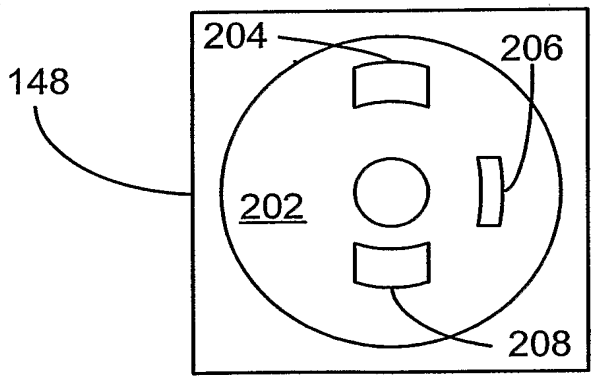

204
206
148
202
208

FIG. 2

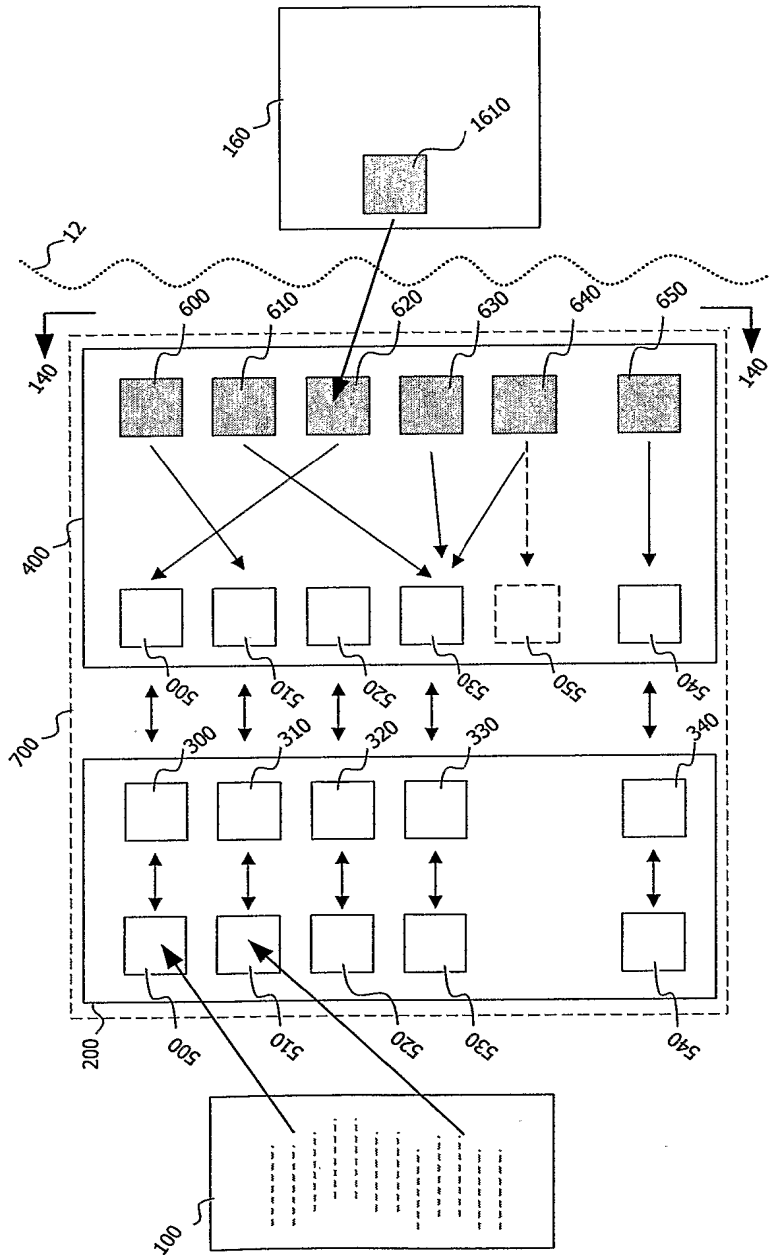

160
1610
12
600
610
620
630
640
650
140
140
400
500
510
520
530
550
540
700
300
310
320
330
340
200
500
510
520
530
540
100

FIG. 3

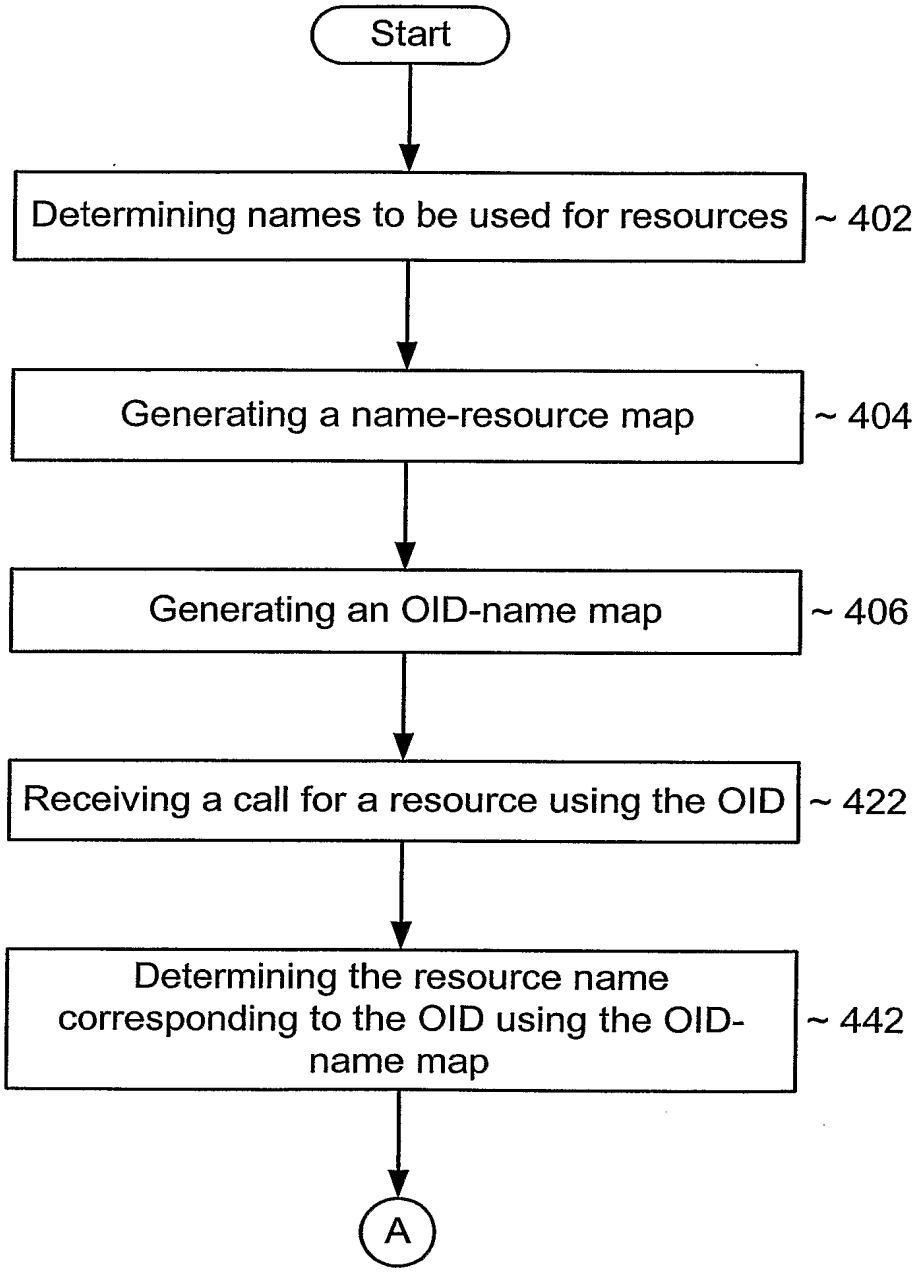

Start
Determining names to be used for resources
~ 402
Generating a name-resource map
~ 404
Generating an OID-name map
~ 406
Receiving a call for a resource using the OID
~ 422
Determining the resource name corresponding to the OID using the OID-name map
~ 442
A

FIG. 4